A Framework for Hybrid Parallel Flow Simulations with a Trillion Cells in Complex Geometries

Christian Godenschwager*†, Florian Schornbaum*, Martin Bauer*,
Harald Köstler* and Ulrich Rüde*

*Friedrich-Alexander-Universität Erlangen-Nürnberg, Cauerstraße 11, Erlangen, Germany
Email: {florian.schornbaum,martin.bauer,harald.koestler,uli.ruede}@fau.de

†Siemens AG, An der Lände 1, 91301 Forchheim
Email: christian.godenschwager.ext@siemens.com

ABSTRACT

waLBerla is a massively parallel software framework for simulating complex flows with the lattice Boltzmann method (LBM). Performance and scalability results are presented for SuperMUC, the world's fastest x86-based supercomputer ranked number 6 on the Top500 list, and JUQUEEN, a Blue Gene/Q system ranked as number 5.

We reach resolutions with more than one trillion cells and perform up to 1.93 trillion cell updates per second using 1.8 million threads. The design and implementation of waLBerla is driven by a careful analysis of the performance on current petascale supercomputers. Our fully distributed data structures and algorithms allow for efficient, massively parallel simulations on these machines. Elaborate node level optimizations and vectorization using SIMD instructions result in highly optimized compute kernels for the single- and two-relaxation-time LBM. Excellent weak and strong scaling is achieved for a complex vascular geometry of the human coronary tree.

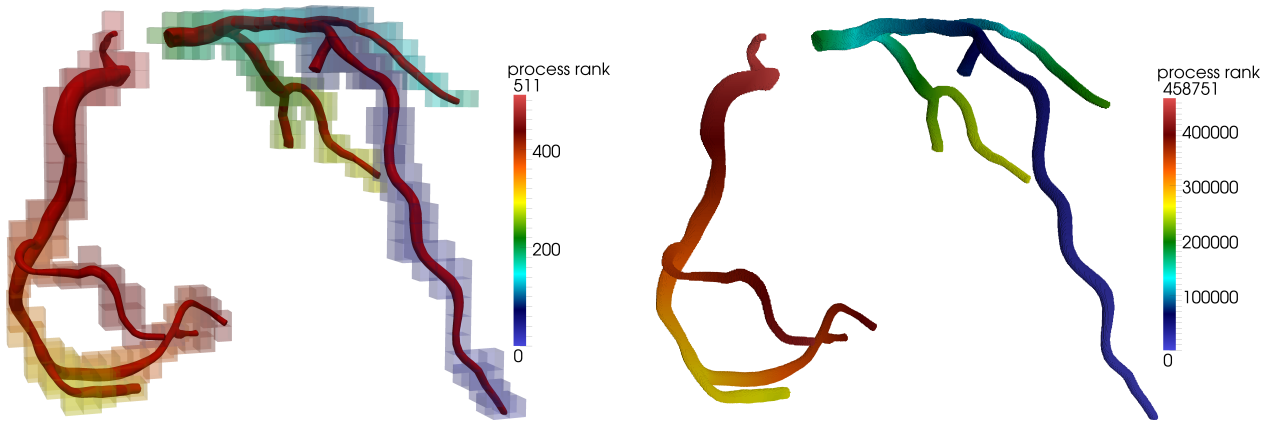
1. INTRODUCTION

Current petascale supercomputers do not only provide unprecedented computing power, they also allow simulations at resolutions that have not been possible before: For our CFD simulations based on the lattice Boltzmann method, storing the data for one trillion cells requires around 277 TiB of main memory. This amount of memory is only available on the largest supercomputers today. However, in order to run efficient, massively parallel simulations that use all cores available on these machines, one meets many challenges. All algorithms, including the initial setup of complex geometries, must run fully in parallel. The initial, static load balancing that evenly distributes all cells to all available compute processes must be able to handle large numbers of cells ($> 10^{12}$). In order to achieve the best performance possible, several levels of parallelism have to be exploited: intra-core (e.g., SIMD vectorization), intra-node (multithreading), and inter-node (message passing). Furthermore, all data structures, especially the data structure holding all the lattice cells of the entire simulation, must be fully distributed. For simulations on hundreds of thousands of cores, these data structures must not grow with the number of compute processes.

In order to tackle these challenges, we present updates to our C++ software framework waLBerla [8], a code for fluid simulations based on the lattice Boltzmann method (LBM). It is suited for solving particulate flows at high efficiency as demonstrated on 294,912 cores of JUGENE [15], a Blue Gene/P system located at Jülich. This is achieved by coupling waLBerla with our in-house rigid body physics engine *pe* [21]. waLBerla may exploit CPU, GPU, and also heterogeneous CPU-GPU clusters [9, 18]. In this paper, we present further optimizations of our LBM kernels' single node performance and the adaption of their implementation to current hardware architectures: We use SSE instructions on Intel and QPX instructions on Blue Gene/Q systems. Using systematic performance engineering, the performance bounds are reached as predicted by performance models. We can show that in terms of performance our implementation of a two-relaxation-time LBM kernel is on par with our most efficient implementation of a single-relaxation-time LBM kernel. Furthermore, we make use of a completely rewritten block-structured domain decomposition of waLBerla: For its data structure managing the domain decomposition, waLBerla now requires a constant amount of memory per process, regardless of the total number of processes and the shape of the domain. Finally, we evaluated the scalability and the performance of our framework on two current petascale supercomputers: the world's fastest x86-based supercomputer SuperMUC located in Munich and Europe's fastest supercomputer JUQUEEN, a Blue Gene/Q system located in Jülich. The motivation was to explore the suitability of Europe's fastest supercomputers for large-scale CFD problems with the LBM and also to compare two dominating HPC architectures.

We show how waLBerla can handle complex geometries described by surface meshes and how simulations are set up with over a trillion lattice cells in a completely parallelized

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC13 November 17-21, 2013, Denver, CO, USA
Copyright is held by the owner/author(s).
Publication rights licensed to ACM.
ACM 978-1-4503-2378-9/13/11...$15.00.
http://dx.doi.org/10.1145/2503210.2503273



(a) One JUQUEEN nodeboard (512 processes, 485 blocks) (b) Whole JUQUEEN (458752 processes, 458184 blocks)

Figure 1: Domain partitioning of a coronary tree dataset with a target of one block per process.

initialization phase. In our block-structured approach, complex geometries introduce load imbalances as blocks may be covered by the computational domain to a different extend. Therefore, we use adaptive mechanisms for domain partitioning and load balancing to the requirements of complex geometries.

Complex geometries are often found in medical applications. Whole frameworks are being developed for computational hemodynamics using the lattice Boltzmann method [27]. Recent comparisons between measurement of fractional flow reserve [31], an invasive method to determine the impact of stenoses in coronary arteries on the heart function, to non-invasive methods based on medical imaging and simulation find that the new diagnostic method shows great promise in classifying coronary lesions with a significantly lower risk and discomfort for the patient compared to invasive methods [25].

Blood flow simulations on the coronary artery tree have been performed with the LBM before. Peters et al. [30] present a massively parallel coupled fluid-particle simulation incorporating red blood cells. They report domains of a billion lattice cells and a spatial resolution of 10 µm similar to a red blood cells diameter of about 7 µm. Bernaschi et al. [4] show similar results utilizing compute clusters with GPUs. Recent petascale machines and our block-structured approach allow us performance experiments with one thousand times as many lattice cells. We report performance results of weak scaling experiments on a geometry extracted from a computed tomography angiography (CTA) dataset of a human coronary artery tree with up to a trillion fluid lattice cells and spatial resolutions down to 1.28 µm, which is less than a fifth of a red blood cell's diameter.

Regarding related work, performance of the block-structured LBM framework OpenLB is reported by Krause et al. [26] but only up to 256 processes. Fietz et al. [10] present methods to improve performance in OpenLB on sparse complex domains by applying an octree-shaped block grid and the shrinking of blocks down to the size overlapped by the computational domain. For complex geometries it is common to use cell-structured LBM approaches with an indirect neighboring scheme different from our block-structured approach. Performance results of cell-structured LBM simulations are reported in [27, 30, 4], while a performance analysis and model is presented by Axner et al. [2]. Groen et al. [17] present performance results for the cell-structured HemeLB framework on vascular geometries on supercomputers HeCTOR and SuperMUC using up to 32,768 cores.

The paper is organized as follows: Section 2 introduces the lattice Boltzmann method, the domain partitioning used by waLBerla, and the methods we use for handling complex vascular geometries in massively parallel simulations. Section 3 describes the two petascale supercomputers SuperMUC and JUQUEEN that we used for our studies. In Section 4, we present our systematic approach of first optimizing the single node performance on these machines, followed by investigating the scalability of our framework for massively parallel runs, and finally showing weak and strong scaling experiments for a simulation of blood flow in the human coronary artery tree. Section 5 concludes the paper.

2. MODELS AND METHODS

Different methods to perform computational fluid dynamics exist. We describe our approach in terms of the underlying model and its implementation.

2.1 The Lattice Boltzmann Method

The lattice Boltzmann method (LBM) is an alternative to classical Navier-Stokes solvers for computational fluid dynamics (CFD) simulations [1]. For 3D LBM simulations, the simulation domain is typically discretized into a uniform Cartesian grid. LBM uses an explicit time stepping scheme that is well suited for extensive parallelization due to its high locality: In each time step, information is only exchanged between neighboring grid cells. In all simulations of this paper we use the three-dimensional D3Q19 model originally developed by Qian, d'Humiéres, and Lallemand [32] with $N = 19$ particle distribution functions (PDFs)

$$f_\alpha : \Lambda \times T \mapsto [0; 1), \tag{1}$$

where $\Lambda \subset \mathbb{R}^3$ and $T \subset \mathbb{R}$ are the physical and time domain, respectively. In general, the LB equation can be written as

$$f_\alpha(x_i + e_\alpha \Delta t, t + \Delta t) - f_\alpha(x_i, t) = \Omega_\alpha(f), \tag{2}$$

with x_i denoting the i-th cell in the discretized simulation domain, e_α denoting the dimensionless discrete velocity set

$\{e_\alpha | \alpha = 0, \ldots, N-1\}$, t denoting the current time step, Δt denoting the time step size, and $\Omega_\alpha(f)$ denoting the LBM collision operator. Algorithmically, the LB equation is typically separated into a collision (3) and a streaming step (4):

$$\tilde{f_\alpha}(x_i, t) = f_\alpha(x_i, t) + \Omega(f_\alpha) \tag{3}$$

$$f_\alpha(x_i + e_\alpha \Delta t, t + \Delta t) = \tilde{f_\alpha}(x_i, t) \tag{4}$$

with $\tilde{f_\alpha}$ denoting the post-collision state of the distribution function. For our studies in this paper, we use two different collision schemes: the single-relaxation-time (SRT/LBGK) model proposed by Bhatnagar, Gross, and Krook [5] and the two-relaxation-time (TRT) model proposed by Ginzburg et al. [13, 12]. The TRT model is more accurate and stable than the SRT model but computationally more expensive. For SRT, the collision operator is given by

$$\Omega_\alpha(f) = -\frac{1}{\tau}(f_\alpha(x_i, t) - f_\alpha^{eq}(x_i, t)), \tag{5}$$

where τ is the relaxation time and $f_\alpha^{eq}(x_i, t)$ is the equilibrium distribution. For TRT, the distribution functions are split into symmetric (even) and asymmetric (odd) parts:

$$f_\alpha^{\pm} = \frac{1}{2}(f_\alpha \pm f_{\bar{\alpha}}) \ , \ f_\alpha^{eq\pm} = \frac{1}{2}(f_\alpha^{eq} \pm f_{\bar{\alpha}}^{eq}), \tag{6}$$

with $\bar{\alpha}$ denoting the inverse direction of α. The corresponding collision operator is given by

$$\Omega_\alpha(f) = \lambda_e(f_\alpha^{+} - f_\alpha^{eq+}) + \lambda_o(f_\alpha^{-} - f_\alpha^{eq-}), \tag{7}$$

with λ_e and λ_o denoting the even and the odd collision parameter of the TRT model. If

$$\lambda_e = \lambda_o = -\frac{1}{\tau}, \tag{8}$$

the TRT model is reduced to SRT. For our simulations, we use no-slip, velocity bounce back, and pressure anti bounce back boundary conditions [14, Chapter 2.5.2].

2.2 The waLBerla Framework

The waLBerla framework employs a block-structured partitioning of the simulation space: Initially, the simulation domain is subdivided into equally sized blocks. The domain can have any arbitrary shape (cf. Figure 2). Each initial block can be further subdivided into eight equally sized, smaller blocks. This process can be applied recursively. The resulting domain partitioning geometrically represents a forest of octrees with each initial block being the root of one octree, similar to the work done by Burstedde et al. [6, 7]. Within each block, a uniform grid of cells is constructed. The grids contained within two different blocks can have different cell sizes, meaning different blocks can posses different grid resolutions. This allows domain partitionings ranging from Cartesian grids that are uniformly split into blocks to octree-like block structures with all blocks containing the same number of cells, thereby forming meshes that resemble grid structures typically associated with grid refinement. Though this is supported in the data structures, our current algorithms and applications do not yet make use of this capability and thus all simulations in this paper use regular grids which are uniformly split into equally sized blocks. Extending our parallel LBM implementation to support blocks with different sizes or grid refinement is subject of current research and will be part of future work.

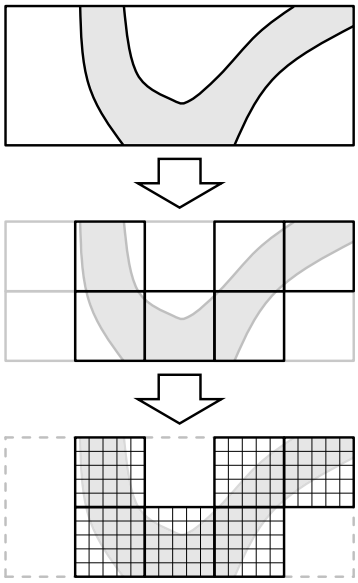


Figure 2: Two-stage domain partitioning: block division and subsequent grid generation.

For MPI parallel simulations, these blocks are distributed among all available processes. Depending on the initial, static load balancing strategy, none, one, or multiple blocks can be assigned to one process. For LBM simulations, the regular grid within each block is extended by one additional ghost layer of cells which is used in every time step during communication in order to synchronize the cell data on the boundary between neighboring blocks. The data structure holding this block structure is fully distributed: Each process only knows about its own blocks and blocks assigned to neighboring processes. Each process only allocates grid and cell data for those blocks assigned to itself. Generally, processes have no knowledge about blocks that are not located in their immediate process neighborhood. As a consequence, the memory usage of a particular process only depends on the number of blocks assigned to this process, and not on the size of the entire simulation, i.e., the total number of blocks assigned to all available processes.

The memory usage during the initialization/construction phase of the block structure (which includes initial, static load balancing) depends on the size of the final simulation and scales linearly with the number of blocks and processes. This initialization, however, can be executed independent from the actual simulation. The resulting block structure can be written to file in order to be later loaded by the simulation. In practice, only one process accesses the file system and loads the entire file into memory using one single read operation. Following this read operation, the binary file content is broadcast to all processes. The file itself is based on a custom endian-independent binary file format which is designed for and heavily optimized towards minimal file size: For simulation variables like process rank or block ID only the lower-order bytes that actually carry information are stored. Even if, for example, storing the process rank requires four bytes of main memory during program execution, only two bytes of disk space are required to store process ranks for simulations with up to 65,536 processes. As a result, block

structures corresponding to simulations with half a million processes can be saved in files that use about 40 MiB of disk space. Consequently, if for large simulations involving hundreds of thousands of processes the per-process memory limit does not allow the initialization phase to be executed during the actual simulation, the block structure can be constructed in advance (even on a different machine with a completely different hardware setup) and loaded from file.

Other frameworks require, at least initially, the entire, fully resolved grid for partitioning [30] and therefore run into problems with very large domains for current, massively parallel supercomputers. In contrast to these frameworks, our hierarchical approach of first dividing the simulation domain into blocks and then later filling these blocks with corresponding parts of the global grid allows us to have computational domains that consist of trillions of cells. Our framework is written entirely in C++ with the code that describes the different LB stencils (D3Q19, D3Q27, D2Q9, etc.) being automatically generated. For low-level parts of the framework we make heavy use of various template programming techniques for achieving high performance while maintaining flexibility. In high-level, non-performance critical parts of the code polymorphism and programming against interfaces ensures flexibility and usability.

2.3 Initialization and Complex Geometries

A property of vascular geometries is the small fraction of space the actual simulation domain covers from the enclosing bounding box. During initialization, we have to decide which blocks are required by the simulation by determining whether they intersect the domain Λ. Furthermore, lattice cells that are either part of the domain or the boundary have to be marked as such. Finally, the blocks have to be distributed among the processes w.r.t. their workload and the communication scheme.

The vascular geometry provides a definition of the domain boundary Γ in form of a triangle surface mesh S. To decide if a point $p \in \mathbb{R}^3$ is within or outside of Λ or whether it lies on Γ, we define an implicit signed distance function

$$\phi(p, \Gamma) = z\mathrm{d}(p, \Gamma) \tag{9}$$

with sign $z \in \{-1, +1\}$ denoting whether p is inside or outside of Γ and $\mathrm{d}(p, \Gamma)$ being the distance of p to Γ given by

$$\mathrm{d}(p, \Gamma) = \mathrm{d}(p, S) = \min_{t \in S} \mathrm{d}(p, t) \tag{10}$$

with t denoting a triangle from S. We calculate $\mathrm{d}(p, t)$ employing the 2D method described by Jones [23]. As proposed by Payne and Toga [29], we reduce computational complexity by subdividing the set of triangles hierarchically into an octree, thus reducing the number of point-triangle distances $\mathrm{d}(p, t)$ actually evaluated. To determine whether p is inside or outside of Λ, i.e., evaluating $\mathrm{d}(p, \Gamma)^2 < 0$, we calculate sign z using the face, edge, and vertex normals of

$$\hat{t}(p) = \arg\min_{t \in S} \mathrm{d}(p, t). \tag{11}$$

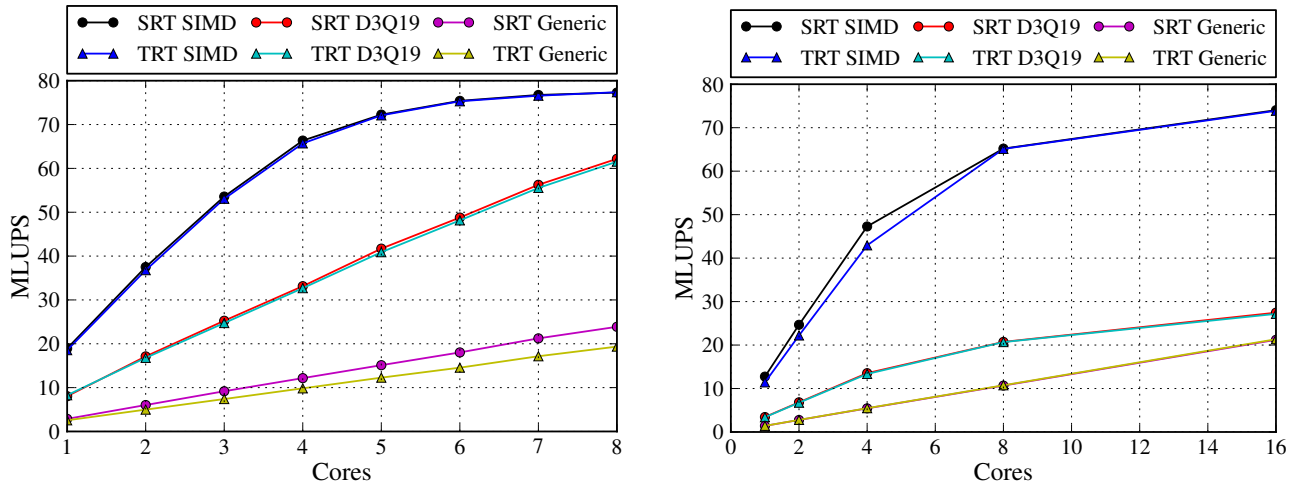
While we consider the face normals of S known, we compute angle weighted pseudonormals for the edges and vertices which guarantee a numerically stable sign computation as shown by Bærentzen and Aanæs [3].

We consider a block b intersecting Λ if the center of any lattice cell in b is within Λ. Again, to reduce computational complexity, we use an octree to avoid testing $\mathrm{d}(x, \Gamma)^2 < 0$ for each lattice cell center x. Furthermore, we compute b's barycenter $\widetilde{b}$, its circumsphere radius $R(b)$, and its insphere radius $r(b)$. If $\mathrm{d}(\widetilde{b}, \Gamma)^2 > R(b)^2$, we can quickly decide that b cannot intersect Λ. Accordingly, if $\mathrm{d}(\widetilde{b}, \Gamma)^2 < r(b)^2$, b must intersect Λ. The process of deciding which blocks are required by the simulation is hybridly parallelized. First all blocks are randomly scattered among the processes to avoid load imbalances, then evaluation takes place as described above. Finally, the result is gathered on all processes.

To perform weak scaling expriments, we seek a domain partitioning yielding a given number of blocks with a fixed block size while varying the isotropic spatial resolution dx. For strong scaling experiments on the other hand, we have to find a fitting block size for a given number of blocks and a fixed dx. We solve both problems by performing a binary search in the respective parameter space, iteratively producing new domain partitionings to match the specified target number of blocks as closely as possible. While for a weak scaling the parameter space is one dimensional by definition, we reduce the dimensionality in the case of a strong scaling by fixing the blocks to cubes and only vary the edge length. As the number of resulting blocks is not monotonic in either the edge length or dx, the result is only a close approximation. Also, due to discretization, an exact solution to the problem may not exist. We therefore use the domain partitioning that yields the most blocks but does not exceed the specified target. If the target implies that every process should be assigned exactly one block, this may lead to a few empty processes, but as shown later in section 4.1, the performance impact of some unused cores is negligible due to the memory bound nature of the LBM code.

Domain partitionings may contain blocks that are completely filled with fluid cells and those which are only partially filled but none which contain no fluid cells at all. As we use LBM kernels that can profit from blocks which are only partially filled by skipping non-fluid cells, we employ load balancing to reduce workload peaks. We assign each block the number of its fluid cells as workload and assign weights to the communication graph that are proportional to the amount of data transferred between neighboring processes. Load balancing has to consider that the memory limit of each process may not be exceeded and that blocks on one process are ideally neighboring each other to exploit fast local communication and save available bandwidth. To solve this multi-constrained optimization problem we use the METIS graph partitioner [24].

As the blocks are finally assigned to processes, we need to initialize the simulation. To mark the fluid cells as such, we voxelize S using ϕ with the proposed optimizations. Every process voxelizes its blocks independently, the only communication required is the initial broadcast of S, which is read by a single process from file. To determine which lattice cells are boundary cells, we compute the hull of the fluid cells using a morphological dilation operator w.r.t. the LBM stencil. To assign specific boundary conditions to the boundary lattice cells, we exploit that S may store a color for each vertex. If the inflow and outflow surfaces of the mesh are unambiguously colored, we can assign each boundary lattice cell a boundary condition according to the vertex colors of the closest triangle $\hat{t}$.



(a) SuperMUC socket

(b) JUQUEEN node

Figure 3: Comparison of the performance of our LBM kernels. Only computation times were measured, communication times were neglected. We used one process per core and SSE on SuperMUC and 4 processes per core and QPX on JUQUEEN (4-way SMT).

3. SUPERCOMPUTER ARCHITECTURES

Our simulations presented in this paper were run on two current petascale supercomputers: the Blue Gene/Q system JUQUEEN located at the Jülich Supercomputing Centre in Jülich and the Intel Xeon-based SuperMUC located at the Leibniz Supercomputing Centre in Munich.

3.1 JUQUEEN

JUQUEEN is a 28 rack Blue Gene/Q system providing 458,752 PowerPC A2 processor cores, with each core capable of 4-way multithreading [11]. These cores run at 1.6 GHz and are designed to achieve low power consumption. A single node of Blue Gene/Q consists of 16 compute cores that deliver up to 204.8 GFLOPS. This leads to a theoretical peak performance for JUQUEEN of 5.9 PFLOPS. For each core, 1 GiB of main memory is available, which sums up to a total of 448 TiB. JUQUEEN uses a 5-dimensional torus network topology capable of achieving up to 40 GB/s, with latencies in the range of a few hundred nanoseconds up to 2.6 µs. Based on these properties and based on our observations on Blue Gene/P [15], we expect our LBM MPI communication to scale to the entire machine.

As of November 2012, JUQUEEN is ranked fifth in the Top500 List [35]. Due to its low power consumption per FLOP, JUQUEEN is also listed on rank 5 in the Green500 List, which provides a ranking of the most energy-efficient supercomputers [16]. Our framework was compiled using the IBM XL compiler in version 12.1 with the compiler optimization level set to 5 (this includes interprocedural optimization at link time).

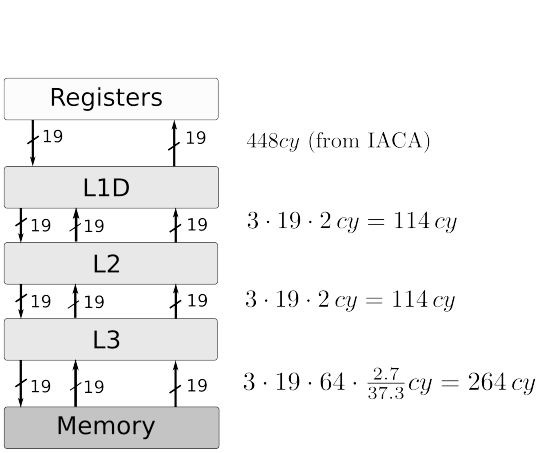
3.2 SuperMUC

The SuperMUC system consists of fewer, but more powerful, cores than JUQUEEN. It is built out of 18432 Intel Xeon E5-2680 processors running at 2.7 GHz, which sums up to a total of 147,456 cores [34]. One compute node of SuperMUC consists of 2 sockets, each equipped with 8 cores. A single node can access 32 GiB of main memory, resulting in 2 GiB per core and 288 TiB for the entire machine. 512 nodes are divided into one island. Within each island, SuperMUC uses a non-blocking tree network topology, whereas all 18 islands are conected via a pruned tree (4:1) [34]. Because of this network topology, we expect to see a drop in the parallel efficiency of our LBM MPI communication when scaling up to multiple islands.

With a peak performance of 3.2 PFLOPS, SuperMUC is the world's fastest x86-based supercomputer, ranked sixth in the Top500 List [35] as of November 2012. On SuperMUC, we used the Intel compiler in version 13.1 together with IBM MPI, which showed slightly better performance than Intel MPI which is also available. For compilation, the optimization level was set to 3 and interprocedural optimization at link time (`-lto`) was enabled. On both machines, compiling our applications with GCC does not yield the same level of performance as compiling with the Intel or IBM compiler, respectively.

4. PERFORMANCE RESULTS

The performance of LBM is best measured in MLUPS or MFLUPS [37], which stands for "million lattice cell updates per second" and "million *fluid* lattice cell updates per second", respectively. When measuring MFLUPS, only fluid cells that are actually processed by the compute kernel are counted. For the MLUPS measurement, not only these fluid cells but all cells that are traveresed during computation, including non-fluid, boundary cells, are considered. We calculate and compare MLUPS when measuring the single node performance of LBM compute kernels on dense grids. If a simulation effectively only consists of fluid cells, we also use MLUPS for simplicity. For all other scenarios, including the sparse vascular geometry of the human coronary tree, we calculate and compare MFLUPS. When showing the performance of scaling experiments we report MLUPS/MFLUPS per core values, which correspond to parallel efficiency.



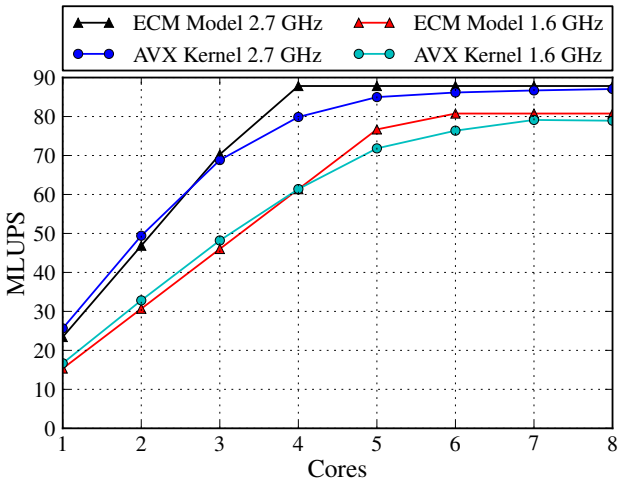


Figure 4: Schematic of the ECM Model for our TRT kernel at a clock frequency of 2.7 GHz (left). Comparison of the ECM model and our measurements for two clock frequencies on SuperMUC (right).

4.1 Single Node Performance

In this section, we present performance values for a single JUQUEEN node and a single socket on SuperMUC. In order to be comparable to literature, all evaluations on SuperMUC were done on one socket instead of doing them on both sockets of the node. The estimated performance values for a single node are twice the socket values, since the computing power and also the bandwidth doubles when scaling from one to two sockets.

Before optimizing the node level performance of the LBM kernel, we utilize the roofline performance model [38] to get an upper-bound on performance, assuming that performance is limited by main memory bandwidth. In the following, we will show that this assumption holds true. To update one fluid cell, 19 `double` values have to be streamed from memory and back. Assuming a write allocate cache strategy and a `double` size of 8 bytes, a total amount of 456 bytes per cell has to be transferred over the memory interface. We determine the maximum attainable socket bandwidth using STREAM [28], resulting in a bandwidth of approximately 40 GiB/s on SuperMUC. A more refined stream benchmark that takes the LBM memory access pattern of multiple concurrent load and store streams into account yields a slightly lower bandwidth of 37.3 GiB/s. Using these values we can estimate an upper performance bound on SuperMUC using the roofline model:

$$37.3\,\frac{\text{GiB}}{\text{s}} \;:\; 456\,\frac{\text{B}}{\text{LUP}} = 87.8\,\text{MLUPS}$$

The same study on JUQUEEN results in a STREAM bandwidth of 42.4 GiB/s (same as reported in [20]). We, however, measure a reduced bandwidth of 32.4 GiB/s when concurrent store streams are used, similar to the memory access patterns in our LBM kernels. Consequently, the roofline model predicts 76.2 MLUPS of theoretically attainable performance on JUQUEEN. To achieve this performance, utilizing the available 4-way simultaneous multithreading (SMT) capabilities of the hardware is crucial (Figure 5), whereas on SuperMUC no performance gain could be measured by using SMT.

We optimize our LBM kernel until the theoretically achievable performance limit is reached. The starting point is a stream-pull kernel, written generically for arbitrary lattice models. A lattice model consists of neighborhood information including a weight for each neighbor. The lattice model is provided to the LBM kernel as a template parameter such that model specific information is already available at compile time. This generic kernel is a naive, textbook-style implementation of the LB method, very similar to the mathematical formulation. The next step is another kernel written specifically for the D3Q19 LB model, enabling the reduction of floating point operations by fusing the streaming and collision step and eliminating common subexpressions in the macroscopic value calculation.

The lattice data structure can be stored in an "Array of Structures" (AoS) or in a "Structure of Arrays" (SoA) layout, which leads to either all PDFs of the same cell or all PDFs of the same direction to be stored consecutively in memory. To make use of the SIMD capabilities of modern architectures, the SoA layout was chosen. Using this layout usually results in too many concurrent load and store streams for the prefetcher to handle. Splitting up the innermost loop of the LBM compute kernel and performing the LBM update in a by-direction rather than a by-cell manner significantly reduces the number of concurrent load/store streams [19]. Performing this complex code transformation for arbitrary lattice models couldn't be done automatically by any of the compilers used during our studies. This code transformation was applied manually in all of our fastest LBM kernels.

The results of these optimizations can be seen in Figure 3. It shows performance results for the three optimization stages of the LBM kernels, each in its SRT and TRT variant. To be comparable to the roofline model, only the time spent in the kernel was taken into account, i.e., communication times are neglected. The results are similar on both machines: As expected, the general kernel is the slowest, followed by the kernel tuned to the D3Q19 model. The SIMD vectorized kernel using AVX on SuperMUC outperforms the non-vectorized version by 20 %, whereas the QPX vectorized JUQUEEN implementation is even 2.5 times faster compared to the

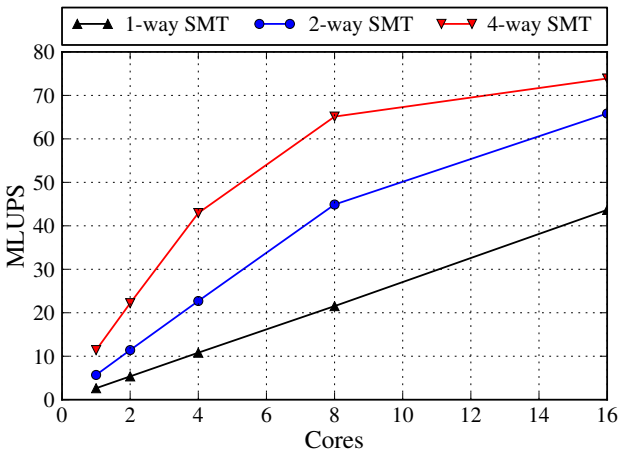


Figure 5: Performance of our optimized TRT LBM kernel on a single node of JUQUEEN using different levels of simultaneous multithreading (SMT).

serial kernel. Despite the increased complexity of the TRT kernel, it is as fast as the SRT kernel on a single node. As shown above, the LBM kernels are memory bound so that the higher FLOP counts of the TRT kernel do not decrease performance. For smaller core counts, however, where the memory interface is not saturated yet, one can see that the TRT kernel is slightly slower than the SRT version. Since the vectorized kernels for both models show the same performance on the full node, we use the TRT version for all simulation runs. The graphs also show that the generic and even the D3Q19 specific kernel are not memory bound on both machines. SIMD vectorization is needed to saturate the memory interface and come close to the predicted limit of the roofline model.

On one SuperMUC compute node, the memory interface can be saturated using only six of the eight cores available on each socket. This facilitates the possibility to reduce energy consumption by choosing a lower CPU clock frequency. The new clock frequency has to be chosen such that all eight cores of the socket are required to saturate the memory interface. In order to predict the dependence of the performance on clock frequency and number of cores, a more refined model is required. The roofline model does not account for the increase in performance scaling from one core to the full socket. It neglects data transfer times inside the cache hierarchy and assumes that the full memory bandwidth can be saturated by a single core. The Execution-Cache-Memory (ECM) model [36, 19] is more general and avoids these limiting assumptions. All levels of the cache hierarchy and the transfer times between them are explicitly modeled. The contributions to the total runtime are divided into three parts: The first contribution is the execution time of the code assuming all required data resides in L1 cache. The second part are the transfer times between caches, and the last part is the time to transfer data over the memory interface. All data transfers are accounted in units of cache lines. One cache line on SuperMUC's Sandy Bridge architecture contains eight `double` values. As a consequence, to employ the ECM model, we need the number of cycles required to perform eight lattice cell updates. We use the Intel Architecture Code Analyzer (IACA) [22] to perform a static analysis of the innermost kernel loops. IACA reports 448 cycles in case of all required data being located in L1 cache. Transferring one cache line between two caches takes two cycles, resulting in a total of 114 cycles for eight lattice cell updates (Figure 4a), assuming a write-allocate cache strategy which leads to three load/store streams in total. Following [36], we assume that each cache can either evict or reload cache lines, but not both at the same time (no-overlap). To estimate the transfer time from/to memory we use the same bandwidth as determined for the roofline model, converted into clock cycles per cache line. On the Sandy Bridge architecture, the main memory bandwidth decreases slightly at lower clock frequencies [33], therefore a stream benchmark was executed for each analyzed clock frequency. Using these inputs, the ECM model suggests an optimal clock frequency of 1.6 GHz, at which 25 % less energy is consumed and still 93 % of the performance can be achieved (Figure 4). The performance penalty of 7 % is due to slightly slower bandwidths at lower clock speeds. For further details on applying the ECM, we refer to [36, 19].

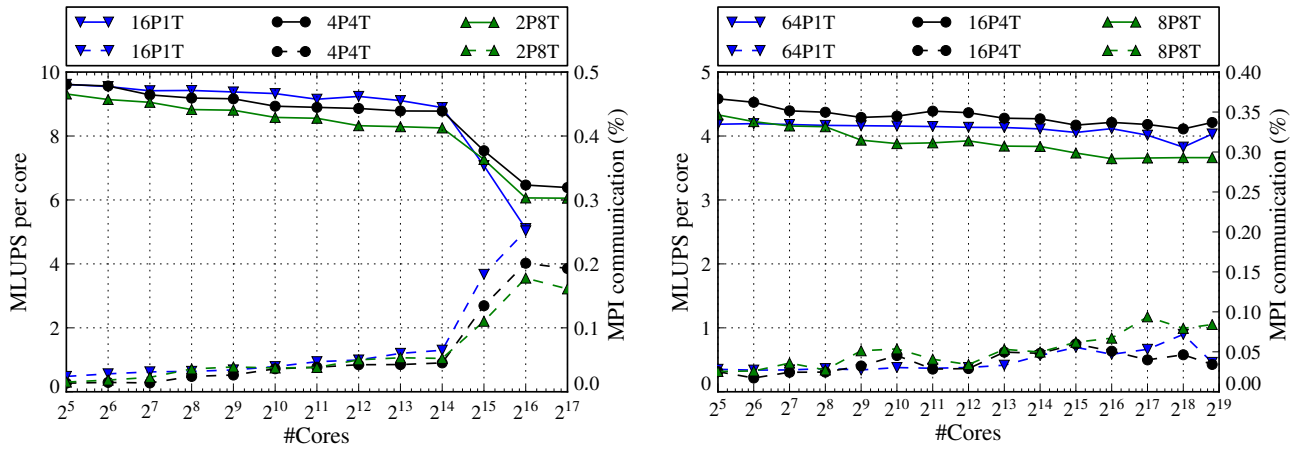
4.2 Dense, Regular Domains

For any software framework to exhibit good scalability on current petascale supercomputers it is crucial that all data structures and algorithms are fully distributed. On the one hand, if the size of any data structure scales with the number of processes, massive parallelization is limited by the growing per-process memory requirement. On the other hand, if the execution time of any algorithm increases with the number of processes, massive parallelization is not possible due to the resulting runtime overhead. In order to test the scalability of our framework, we performed weak scaling experiments with two simple scenarios: the lid-driven cavity problem and channel flow around a fixed obstacle with an obstacle to fluid ratio of less than 1 %. For all measurements, our fastest TRT LBM kernels were used. These scenarios should scale perfectly if communication latency and bandwidth between every pair of neighboring processes remain constant when scaling up the simulation. Due to the various network topologies of different supercomputers, this is typically not the case. We expect our parallel efficiency to be determined by the performance of the communication network.

On SuperMUC (see Figure 6a), we compare three different versions of our framework: one pure MPI parallelization (16 processes per node) and two MPI/OpenMP hybrid versions (4 processes per node with 4 threads per process and 2 processes per node with 8 threads per process). When scaling up, we keep the number of cells per core constant at 3,430,000, resulting in 4.5×10^{11} cells for the largest run. We achieve up to 837×10^3 MLUPS, meaning 837 billion cells are updated per second, which corresponds to 166 TFLOPS or about 5 % of peak performance. This is due to LBM being limited by the memory bandwidth as shown in the previous section. However, with 347 TiB/s of aggregated bandwidth, we reach 54.2 % of the total memory bandwidth theoretically available for 2^{17} cores:

$$\frac{837 \cdot 10^9 \cdot 19 \cdot 3 \cdot 8 : 1024^3 \text{ GiB/s}}{2^{14} \cdot 40 \text{ GiB/s}} = 54.2\,\%$$

The framework manages to scale up to 2^{17} cores without any overhead generated by its data structures or algorithms. We, however, see that the percentage of time spent for MPI communication increases the more processes we use. This increase in MPI communication time corresponds to the



(a) SuperMUC: 3,430,000 lattice cells per core

(b) JUQUEEN: 1,728,000 lattice cells per core

Figure 6: Weak scaling on SuperMUC and JUQUEEN: αPβT means α processes per node with β threads per process. Solid lines show MLUPS per core values for a given setup. Dotted lines indicate the corresponding percentage of time spent for MPI communication. For all measurements, TRT LBM kernels were used.

drop in parallel efficiency. Consequently, for weak scaling experiments the parallel efficiency of our framework is only limited by the communication network of SuperMUC.

On JUQUEEN (see Figure 6b), we also compare three different versions: one pure MPI parallelization (64 processes per node) and two MPI/OpenMP hybrid versions (16 processes per node with 4 threads per process and 8 processes per node with 8 threads per process). Due to less memory per core on JUQUEEN, we can only assign 1,728,000 cells to each core, which still results in 7.9×10^{11} cells for the largest run. During this simulation, more than 1.8 million threads manage to update 1.93 trillion cells per second, which corresponds to 383 TFLOPS or about 6.5 % of peak performance. Again, this is due to LBM being bandwidth limited. However, with 800 TiB/s of aggregated bandwidth, we reach 67.4 % of the total memory bandwidth theoretically available on the entire machine:

$$\frac{1.93 \cdot 10^{12} \cdot 19 \cdot 3 \cdot 8 : 1024^3 \,\text{GiB/s}}{458\,752 : 16 \cdot 42.4 \,\text{GiB/s}} = 67.4\,\%$$

Again, parallel efficiency is determined by the scalability of the communication network. Since the percentage of time spent for MPI communication is quite stable when scaling up to the entire machine, we can achieve a parallel efficiency of 92 % for all 458,752 cores of JUQUEEN.

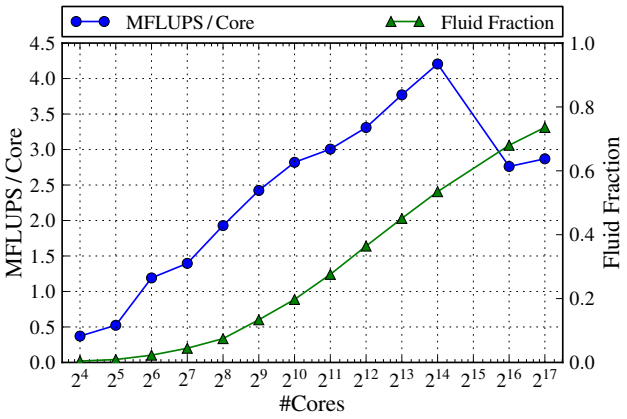
4.3 Complex Vascular Geometry

To evaluate the performance of waLBerla on complex geometries, we performed simulations on a human coronary tree. The geometry was extracted from a computed tomography angiography dataset. As this geometry only covers about 0.3 % of the volume of its enclosing axis-aligned bounding box, it presents a challenge for our block-structured LBM approach. Blocks may be only partially covered by the computational domain and superfluous lattice cells which are neither boundary nor fluid may be generated.

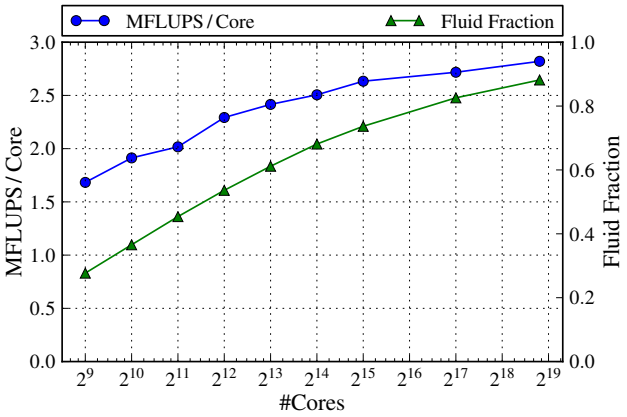
We use different strategies in our LBM kernels to maintain high performance even if a block is only partially covered with fluid lattice cells. The simplest strategy is to introduce a conditional statement and execute the LB stream and collide steps only if the active lattice cell is a fluid lattice cell. While introducing this conditional statement in the innermost kernel loop induces a major performance penalty in itself, it is also incompatible with vectorization. Another approach is to store the coordinates of a block's fluid lattice cells in an array and loop over this array instead of all lattice cells of the block. This removes the conditional statement from the kernel, but still vectorization is impossible. Finally, we store for every line of lattice cells the index of the first and last fluid lattice cell, similar to the compressed storage scheme of a sparse matrix. We then execute our kernel only on the lattice cells between those indices. This approach enables vectorization and fits our geometry as it has few but consecutive fluid lattice cells. Nevertheless, we expect a negative performance impact of sparse blocks, as some of the skipped lattice cells may be loaded by the prefetcher unit anyways. Additionally, our communication scheme is unaware of fluid lattice cells and therefore the amount of data communicated between neighboring blocks is the same as for densely populated blocks.

Considering our experiences on dense, regular domains (see Figure 6), we performed experiments on JUQUEEN using 16 processes per node with four threads per process. On SuperMUC, we ran simulations with four processes per node and four threads per process. To achieve a similar number of accumulated fluid lattice cells per process, we allocate up to four blocks on every process and enable load balancing. For each data point we measured, we collected at least ten samples with a minimum of six time steps each. In Figures 7 and 8 we report the maximum values.

On SuperMUC, we performed a weak scaling up to 131,072 cores (Figure 7a) using blocks of size 170^3 (19.6 million lattice cells per process). This corresponds to spatial resolutions from 0.1837 mm (1 node) down to 1.5274 µm on 16 of the 18 islands available on the machine. On JUQUEEN, we scaled up to the full machine with 458,752 cores (Figure 7b) using blocks of size 80^3 (2,048,000 lattice cells per process). The spatial resolution varied from 0.0180 mm (1 nodeboard) down



(a) SuperMUC



(b) JUQUEEEN

Figure 7: Results of weak scaling experiments with the complex vascular geometry. Due to our block-structured implementation, we see an increase in efficiency as the fraction of fluid lattice cells also increases.

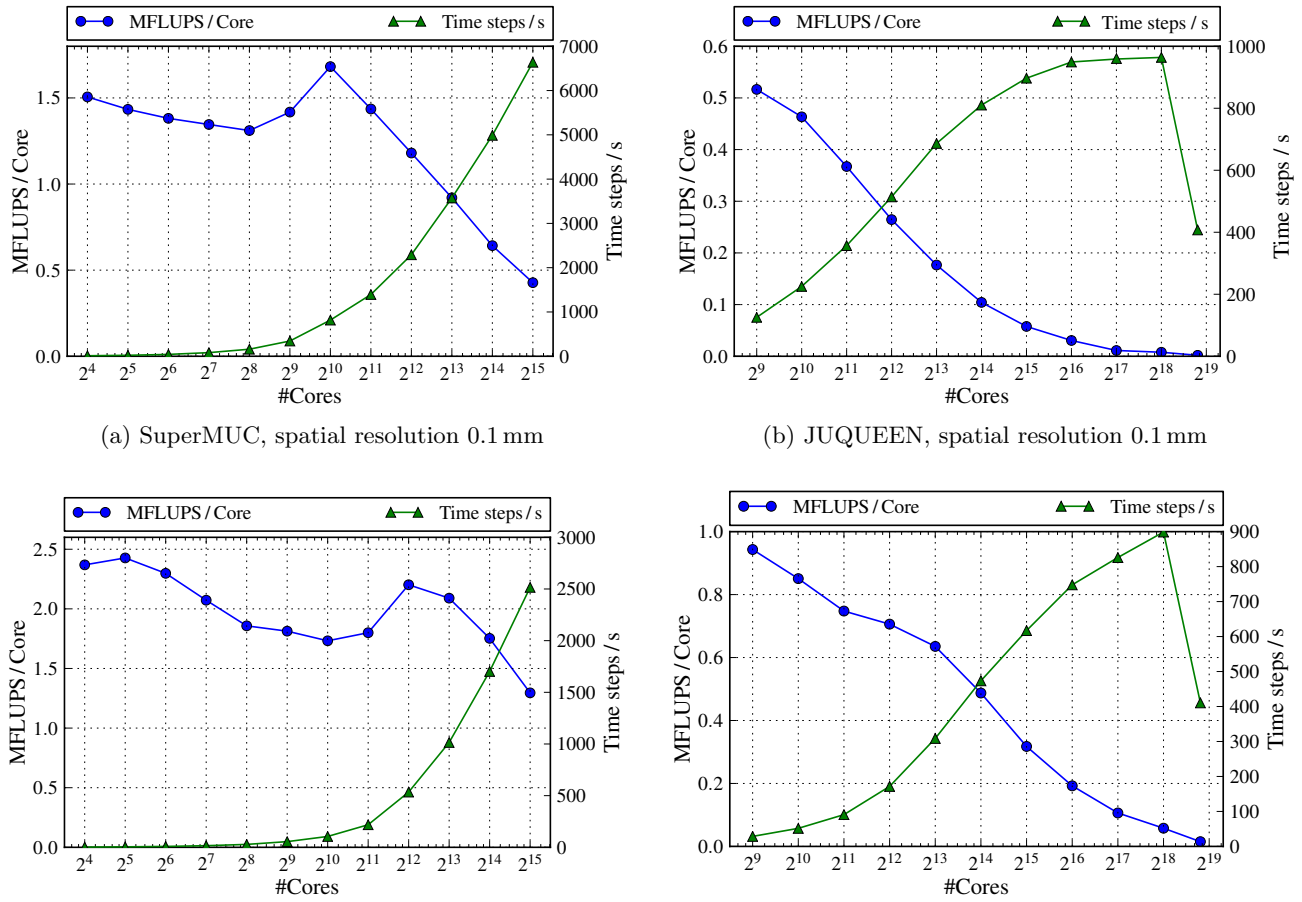
to 1.276 µm on the whole machine. This is less than one fifth of a typical red blood cell's diameter. As we expected, results show an increase in MFLUPS / core with an increasing number of cores used instead of a nearly constant behavior as reported in section 4.2. Using more processes results in using more blocks which present a better fit to our geometry (cf. Figure 1). This effect manifests in a larger fraction of fluid cells from which the efficiency of both our compute kernels and communication can benefit due to the reasons mentioned in the description of the compute kernels earlier in this section. An exception are the large scale runs on SuperMUC, where we see the same drop in performance caused by increased time spent in communication as described in section 4.2. Our weak scaling results show that waLBerla with its block-structured approach is well suited to perform efficient simulations of coronary blood flow, especially on large scales. Particularly, at very large resolutions, which are hard to achieve with cell-structured approaches due to load balancing and memory considerations, we show excellent efficiency with a cell size at just a fraction of a red blood cell's diameter.

While the study of blood flow on a rather microscopic level is an obviously well suited problem for supercomputers to solve, many applications use a more macroscopic approach and are content with lower resolutions. As the LBM is an explicit time stepping scheme the time step has to decrease with an increasing spatial resolution. Considering that our method is stable up to a lattice velocity of 0.1 and assuming a maximal blood velocity of 0.2 m/s, the time step length computes to half the spatial resolution. For a spatial resolution of 1.276 µm we have a time step length of 0.64 µs and achieve 1.25 time steps per second using 458,752 cores on JUQUEEN. For most production runs, we are interested in waLBerla's performance on smaller problems and in the scales at which a supercomputer may be efficiently used to solve such a problem. Therefore, we conducted strong scaling experiments with resolutions of 0.1 mm (2.1 million fluid lattice cells in the entire domain) and 0.05 mm (16.9 million fluid lattice cells in the entire domain).

For strong scaling experiments, we partition rather small domains into a large number of blocks. Consequently, the larger the number of blocks the fewer lattice cells are contained within each block. On the one hand, smaller blocks can favor performance due to the domain partitioning providing a better fit to the geometry. On the other hand, smaller blocks result in less time spent in the optimized LBM kernel and more time in communication and our framework's code to manage control flow. Accordingly, we conducted the strong scaling experiments with varying numbers and varying sizes of blocks. In Figure 8, we report the maximum performance achieved.

Results on SuperMUC show a rather stable efficiency up to a certain number of nodes (128 nodes at 0.1 mm and 512 nodes at 0.05 mm resolution) and a decline afterwards. The decline is caused by the blocks getting to small to be efficiently handled by our framework. At 0.1 mm resolution, performance increases continuously from 11.4 time steps / s on a single node up to 6638.31 time steps / s on 2048 nodes. The optimal number of blocks per core is 32 at 16 cores declining to 1 at 4,096 cores. Block sizes range from 34^3 at 16 cores down to 9^3 on 32,768 cores. At a resolution of 0.05 mm, we achieve a better baseline performance of 2.25 time steps / s on a single node which increases up to 2515.58 time steps / s on 2048 nodes. The optimal number of blocks per core is 64 at 16 cores down to 1 at 16,384 cores. Block sizes range from 46^3 to 13^3. Consequently, SuperMUC may be used efficiently up to a fair degree of parallelism. Targeting short time-to-solution may allow for an even larger degree of parallelism, but at the cost of wasting valuable compute resources due to decreasing efficiency at larger scales.

On JUQUEEN, we see a different picture (Figure 8c and Figure 8a). Efficiency decreases consecutively from a baseline of 0.51 MFLUPS / core (0.1 mm resolution) and 0.94 MFLUPS / core (0.05 mm resolution) at the smallest accountable unit of one nodeboard. Optimal number of blocks per core and block sizes are similar to SuperMUC's at the respective job sizes. We find that the architecture of SuperMUC's Intel processors with more FLOPS / core than IBM's PowerPC processors are better suited to deal with the framework overhead exposed by very small block sizes. Nevertheless, performance on JUQUEEN rises up to the use of 65,536 cores (0.1 mm resolution) and 262,144 cores (0.05 mm resolution). The decreasing efficiency does not allow for values similar to SuperMUC's, though.



(a) SuperMUC, spatial resolution 0.1 mm

(b) JUQUEEN, spatial resolution 0.1 mm

(c) SuperMUC, spatial resolution 0.05 mm

(d) JUQUEEN, spatial resolution 0.05 mm

Figure 8: Performance results of strong scaling experiments using a complex vascular geometry.

5. CONCLUSION

The LBM method implemented in waLBerla runs efficiently on current petascale supercomputers and shows excellent scaling properties. Our performance measurements agree well with the predictions that are obtained from detailed performance models, as they have been developed within an integrated performance engineering methodology. This demonstrates also that we reach close to optimal efficiency, both on the node level, as well as for the whole system. Thus waLBerla reaches execution rates of up to 6638 LBM time steps per second in strong scaling setting and with a resolution that could be used for practical real-time simulations. In the largest weak scaling simulations, a human coronary artery tree has been discretized with a record of 1,033,660,569,847 fluid lattice cells. This corresponds to a resolution of 1.276 µm which is less than one fifth of the size of a typical red blood cell. Thus simulations of a complete tree of blood vessels are possible down to a mesh resolution well below the size of a biological cell. To the best of our knowledge, such fine resolutions and such large ensembles have not been previously reported for LBM blood flow simulations. In future work, we will extend waLBerla to support blocks with different sizes and grid refinement, permitting better resolution in critical subdomains. This will also require dynamic load balancing and will be especially useful for applications such as particulate flows with fully resolved particles and moving geometries.

6. ACKNOWLEDGMENTS

We are grateful to the Jülich Supercomputing Center and the Leibniz Rechenzentrum in Munich for providing computational resources.

7. REFERENCES

[1] C. K. Aidun and J. R. Clausen. Lattice-Boltzmann method for complex flows. *Annual Review of Fluid Mechanics*, 42:439–472, 2010.

[2] L. Axner, J. Bernsdorf, T. Zeiser, P. Lammers, J. Linxweiler, and A. Hoekstra. Performance evaluation of a parallel sparse lattice Boltzmann solver. *Journal of Computational Physics*, 227(10):4895 – 4911, 2008.

[3] J. Bærentzen and H. Aanæs. Signed distance computation using the angle weighted pseudonormal. *Visualization and Computer Graphics, IEEE Transactions on*, 11(3):243–253, 2005.

[4] M. Bernaschi, M. Bisson, M. Fatica, S. Melchionna, and S. Succi. Petaflop hydrokinetic simulations of

complex flows on massive GPU clusters. *Computer Physics Communications*, 2012.

[5] P. L. Bhatnagar, E. P. Gross, and M. Krook. A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems. *Phys. Rev.*, 94:511–525, May 1954.

[6] C. Burstedde, O. Ghattas, G. Stadler, T. Tu, and L. C. Wilcox. Towards adaptive mesh PDE simulations on petascale computers. *Proceedings of Teragrid*, 8, 2008.

[7] C. Burstedde, L. C. Wilcox, and O. Ghattas. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[8] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde. WaLBerla: HPC software design for computational engineering simulations. *Journal of Computational Science*, 2(2):105 – 112, 2011.

[9] C. Feichtinger, J. Habich, H. Köstler, G. Hager, U. Rüde, and G. Wellein. A flexible Patch-based lattice Boltzmann parallelization approach for heterogeneous GPU-CPU clusters. *Parallel Computing*, 37(9):536–549, 2011.

[10] J. Fietz, M. J. Krause, C. Schulz, P. Sanders, and V. Heuveline. Optimized hybrid parallel lattice Boltzmann fluid flow simulations on complex geometries. In C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 818–829. Springer Berlin Heidelberg, 2012.

[11] M. Gilge. IBM System Blue Gene Solution: Blue Gene/Q Application Development. *IBM Redbook Draft SG24–7948–00*, 2012.

[12] I. Ginzburg, F. Verhaeghe, and D. d'Humieres. Study of simple hydrodynamic solutions with the two-relaxation-times lattice Boltzmann scheme. *Communications in computational physics*, 3(3):519–581, 2008.

[13] I. Ginzburg, F. Verhaeghe, and D. d'Humieres. Two-relaxation-time lattice Boltzmann scheme: About parametrization, velocity, pressure and mixed boundary conditions. *Communications in computational physics*, 3(2):427–478, 2008.

[14] J. Götz. *Massively Parallel Direct Numerical Simulation of Particulate Flows*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, May 2012.

[15] J. Götz, K. Iglberger, M. Stürmer, and U. Rüde. Direct numerical simulation of particulate flows on 294912 processor cores. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[16] Green500. www.green500.org, Mar 2013.

[17] D. Groen, J. Hetherington, H. B. Carver, R. W. Nash, M. O. Bernabeu, and P. V. Coveney. Analyzing and modeling the performance of the HemeLB lattice-Boltzmann simulation environment. *arXiv preprint arXiv:1209.3972*, 2012.

[18] J. Habich, C. Feichtinger, H. Köstler, G. Hager, and G. Wellein. Performance Engineering for the Lattice Boltzmann Method on GPGPUs: Architectural Requirements and Performance Results. *Computers & Fluids*, 2012.

[19] G. Hager, J. Treibig, J. Habich, and G. Wellein. Exploring performance and power properties of modern multicore chips via simple machine models. *CoRR*, abs/1208.2908, 2012.

[20] R. Haring and B. Team. The Blue Gene/Q compute chip. In *The 23rd Symposium on High Performance Chips (Hot Chips)*, 2011.

[21] K. Iglberger. *Software Design of a Massively Parallel Rigid Body Framework*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, Sep 2010.

[22] Intel Coorporation. Intel architecture code analyzer, version 2.0.1, http://software.intel.com/en-us/articles/intel-architecture-code-analyzer/, Jan 2013.

[23] M. W. Jones. 3D distance from a point to a triangle. Technical report, Department of Computer Science, University of Wales, 1995.

[24] G. Karypis and V. Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1992.

[25] B.-K. Koo, A. Erglis, J.-H. Doh, D. V. Daniels, S. Jegere, H.-S. Kim, A. Dunning, T. DeFrance, A. Lansky, J. Leipsic, et al. Diagnosis of ischemia-causing coronary stenoses by noninvasive fractional flow reserve computed from coronary computed tomographic angiograms. *Journal of the American College of Cardiology*, 58(19):1989–1997, 2011.

[26] M. Krause, T. Gengenbach, and V. Heuveline. Hybrid parallel simulations of fluid flows in complex geometries: Application to the human lungs. In M. Guarracino, F. Vivien, J. Träff, M. Cannatoro, M. Danelutto, A. Hast, F. Perla, A. Knüpfer, B. Martino, and M. Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science*, pages 209–216. Springer Berlin Heidelberg, 2011.

[27] M. Mazzeo and P. Coveney. HemeLB: A high performance parallel lattice-Boltzmann code for large scale fluid flow in complex geometries. *Computer Physics Communications*, 178(12):894 – 914, 2008.

[28] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[29] B. Payne and A. Toga. Distance field manipulation of surface models. *Computer Graphics and Applications, IEEE*, 12(1):65–71, 1992.

[30] A. Peters, S. Melchionna, E. Kaxiras, J. Lätt, J. Sircar, M. Bernaschi, M. Bison, and S. Succi. Multiscale simulation of cardiovascular flows on the IBM Bluegene/P: Full heart-circulation system at red-blood cell resolution. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[31] N. H. Pijls, B. de Bruyne, K. Peels, P. H. van der Voort, H. J. Bonnier, J. Bartunek, and J. J. Koolen. Measurement of fractional flow reserve to assess the functional severity of coronary-artery stenoses. *New England Journal of Medicine*, 334(26):1703–1708, 1996. PMID: 8637515.

[32] Y. Qian, D. d'Humieres, and P. Lallemand. Lattice BGK models for navier-stokes equation. *EPL (Europhysics Letters)*, 17(6):479, 1992.

[33] R. Schöne, D. Hackenberg, and D. Molka. Memory performance at reduced CPU clock speeds: an analysis of current x86_64 processors. In *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, pages 9–9. USENIX Association, 2012.

[34] SuperMUC Petascale System. https://www.lrz.de/services/compute/supermuc/systemdescription/, Mar 2013.

[35] Top500. www.top500.org, Mar 2013.

[36] J. Treibig and G. Hager. Introducing a performance model for bandwidth-limited loop kernels. In *Parallel Processing and Applied Mathematics*, pages 615–624. Springer, 2010.

[37] G. Wellein, T. Zeiser, G. Hager, and S. Donath. On the single processor performance of simple lattice Boltzmann kernels. *Computers & Fluids*, 35(8):910–919, 2006.

[38] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.